

2 DirectX and DirectDraw

2.1 Aims

In this Module we get into some serious DirectX action using DirectDraw. The aim of this module is to get you used to the DirectX interface and to introduce the fundamentals of 2D DirectDraw.

2.2 Getting Started with DirectDraw

In this section we'll run through the fundamentals of getting a window or a screen on which you can start doing some drawing.

2.2.1 Creating a DirectDraw Object

`DirectDrawCreate()` covers up most of the process in terms of getting a DirectDraw COM interface. Once you have a basic `IDirectDraw` interface, you can use it to get other interfaces using the `IUnknown` method `QueryInterface()`. This is particularly useful for finding what level of DirectX is supported on the machine you are using, and handling your code appropriately. For example if you request a DirectX 6.0 interface on a machine with DirectX 5.0, `QueryInterface()` will give you an error to which your code can respond by either not using some 6.0 features, or telling the user to get his or her act together and update DirectX. Throughout this section we'll be using the `IDirectDraw4` interface.

2.2.2 Cooperating with Windows

DirectDraw gives you the power of life and death over the video hardware and, surprisingly, a lot of the CPU as well. For full-screen, CPU-munching action this is a good thing, but for a windowed application you really don't need or want all that responsibility. DEMO6_1 illustrates the basics of a cooperative DirectX window. As for our GDI stuff though, you are going to have to look after resizing events and even more carefully in DirectX.

2.2.3 Mode Settings

Anybody who has played with the Windows settings for their video card will be familiar with range of video mode options available to you. New cards now go up to

1600 x 1200 in 32 bit color, with up to 64 Mb of video RAM. Most machines have at least 8 Mb of RAM these days, so there should be no problems creating back buffers at reasonable resolutions for most games these days.

2.2.4 Colour Formats

The colour of a pixel can be expressed as 1, 2, 4, 8, 16, 24 or 32 bits. The 1, 2, 4 and 8 bit formats are lookup of a 24 bit value in a palette of colours as you did in GDI. Palettes are hardly used in current games. Most games use the 16, 24 and 32 bit formats, which encode the red, green and blue components of the colour, with the 32 bit format also having a transparency (alpha) component.

2.2.5 Somewhere to Draw

The software abstraction of the video memory that is currently being displayed on the screen is called the *primary surface*. When you modify the primary surface you are modifying the actual values used by the rasterising hardware to generate the video. Your change of a pixel's value will be reflected on the screen as soon as the rasterising hardware sweeps over that pixel again. Hardware and memory formats vary from graphics card to graphics card, so DirectX does just enough abstraction to make sure that you can rub the right pixel without explicit knowledge of how the memory is organised.

Alternatively, you can draw on *secondary surfaces*. Secondary surfaces are usually in video memory and operate just like a primary surface except that they don't get displayed. With modern AGP cards, surfaces can also exist in system memory without much slow down.

To create a surface, you call `CreateSurface()` (a bit tricky to remember that one). Only three parameters, but unfortunately the first parameter is a nested structure with a bazillion elements. Read up the text for the details. Ultimately you get handed a pointer to somewhere to draw. This pointer actually points to your video memory in whatever size and colour mode you specified.

2.2.6 Plotting Pixels

Once you have a surface, you can now start drawing. Remember though that we are using DirectX so that we can use the video card in the fastest way possible. That means to plot a pixel, you actually need to find the pixel you want in video memory, and then now what the bits mean. This was easy with GDI, because the software had a nicely laid out viewport coordinate system and a pre-defined colour interface. With DirectDraw you need to pay attention to the pixel format to optimise your code to draw as quickly as possible.



This has all been a review of Chapter 6, now moving to the first few pages of Chapter 7 (all of which you should have read). It is also worth having a look at the DirectX Foundations in the MSDN (under Platform SDK / Graphics and MultiMedia / DirectX).

In order to find out the pixel format, you need to call another of those crazy, hard-to-remember methods called `GetPixelFormat()`. `Demo6_3` relies on the fact that the 8-bit standard is pretty ... well ... standard. In practice, you should call `GetPixelFormat()` to make sure your plotting what you think you are plotting.



Modify DEMO6_3 to run in various full-screen modes. Set up hot-keys to change between 640 x 480 8-bit, 640 x 480 32-bit, 1024 x 768 8-bit and 1024 x 768 32-bit. In the 32-bit mode, use all the colour space not just 256 colours. See DEMO7_2 for more hints.

2.3 More DirectDraw

Now you know how to create a DirectDraw surface and perform the most primitive operation – pixel plotting. Let’s now have a look at some of the other DirectDraw features you’ll need for writing games.

2.3.1 Back Buffers and Page Flipping

A back buffer is a piece of video memory that you use for your rendering. The nice thing about back buffers is that you can draw on one while the primary surface is being displayed, without bugging up the primary surface. When you finish drawing, you tell the graphics card that the back buffer is now the primary surface (and vice versa) and you instantly start displaying the updated image. This is how we get rid of the annoying flicker we saw in the GDI exercises. It’s called *page flipping*.

Page flipping is accomplished by changing some pointers in the video card that indicate where the primary surface is. You don’t have to worry about that – you just need to establish a primary surface with a back buffer and call `Flip()` (can’t get over these tricky, tricky names).

DirectX calls a back buffer an attached surface. You create the back buffer when you create the primary surface by fiddling a few parameters. Then call `GetAttachedSurface()` and you have a new place to draw. You can draw in the same way that you drew on the primary surface.

2.3.2 Blitting

Blitting is filling, copying or moving a block of pixels. If you want to move around 2D figures for a game, you’ll be doing a lot of this. Copying pixel by pixel takes a lot of CPU cycles, that I’m sure you’ll find useful for other things. This is where the hardware blitter comes in; and, of course, its DirectX interface.

To blit, call `Blit()`. The good stuff for blitting is hidden in the `dwFlags` parameter, and the lengthy special effect structure `lpDDBltFx`. The DEMO7_6 and DEMO7_7 examples show you how to fill and copy bitmaps respectively.



Replicate the functionality of DEMO4_4 using DirectX. Instead of bouncing an ellipse all over the screen, bounce a square instead. Use blitting and page flipping to make your graphics code run fast and smooth. Run as a full screen application using 800 x 600, 16 bit colour.

2.3.3 Clipping

Clipping is a HUGE issue in graphics programming, and seems like one of those annoying details that you would just like to ignore. However, clipping can be the

make or break of a graphics engine, even more so in 3D. Again, hardware clipping is the key, and DirectX is the key to the hardware.

IDirectDrawClipper is the interface for all 2D clipping. It's methods are straightforward. Once the object is created, you set up a *clipping list*. A clipping list is a list of rectangular regions where drawing *can* occur. You then attach the clipping list to the surface that you want to clip (usually the back buffer), and get on with drawing.



Extend the functionality of your new DirectX DEMO4_4 to have a 100 pixel wide coloured bar that runs from the top-middle of the screen to the bottom-middle of the screen. Use clipping to make your bouncing square appear to move behind the bar. Feel free to use the clipper function in DEMO7_9.

2.3.4 Bitmaps

We'll be working with bitmaps for a lot of our 2D animation. The text recommends you to use separate .BMP files that you can edit with whatever you like. The cost of this is the all the bitmap file reading code you need. Since Andre has kindly supplied this, you can go ahead and work this way. Personally, I find the IDE bitmap editor good enough for most things, so I keep the bitmaps as a resource which can be loaded with `LoadBitmap()`. This also makes naming and organisation easier, so you don't need to worry about templates. If you want to bring in something particularly groovy, you can always import it over the old file. You choose.

2.3.5 Off-screen Surfaces

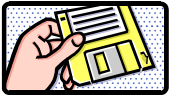
Remember, way back in Section 2.2.5, we talked about secondary surfaces? Well now you can use one for something useful. Because blitting is fastest between video memory surfaces, the best place to keep your bitmaps is on an off-screen secondary surface. Then you can quickly blit your little bitmaps wherever you want on the back buffer, creating you animated effects. The only problem you have now is what to do about blitting a non-rectangular shape.

2.3.6 Colour Keying

Colour keys are special values stored in a bitmap that tell the blitter that these pixels should not be blitted. For example, if you want to blit a bitmap of a round ball on to the screen:

1. create a bitmap with the centre round region all some colour (let's say green),
2. set the other bits as transparent, by assigning them to some colour, and
3. let the blitter know that the transparent colour should not be copied.

This is called source colour keying as the source bitmap is instructing which pixels should be copied. Similarly you can mask out regions of the screen in this way by having destination colour keying, where pixels in the destination bitmap are marked as opaque and not to be copied over. This is handy for making isometric or plan view games where you want characters to disappear behind scenery.



Further extend the functionality of your new DirectX DEMO4_4 to bounce around a round ball like the original DEMO4_4. Load the ball from a bitmap.

2.3.7 Bitmap Rotation and Scaling

Nearly all modern hardware supports scaling, but few support rotation. Change the size of your bitmap by simply blitting with a different destination rectangle size. Change the rotation angle by setting the rotate flag, and specifying the angle of rotation in 1/100 of a degree in the effects structure. You really want your hardware to do this as the software overhead is enormous. It's probably a good idea to do a capability check before trying (see 2.3.9).

2.3.8 GDI in DirectX

GDI in DirectX is disgustingly simple. Grab a GDI device context from DirectDraw using `GetDC()` and GDI to your heart's content.



Change your new DirectX DEMO4_4 to draw the ball on to the secondary surface with GDI, instead of loading it from a bitmap.

2.3.9 Checking Capabilities

`GetCaps()` can let you know what you're system can do in hardware, can do in hardware emulation and cannot do all together. For details of the structures and results check the SDK help.



Check whether your hardware or hardware emulation supports bitmap rotation. If it does, change your DirectX DEMO4_4 to bounce a triangle around the screen. Always point the triangle in the direction of motion.

2.3.10 Windowed DirectX

It's clearly important to make your program behave sensibly when windows are moved, resized, closed etc. This also applies to windowed DirectX programs.



Fix up DEMO7_19 to properly draw inside the window frame. Restore the Minimise, Maximise and Close buttons, and make sure that they work properly.

2.4 Preparation for Next Week



Read Chapter 8, and also the first part of Chapter 9, up to where the text starts to talk about "trapping the mouse". Pages 397 – 566.